Tcl and SDC Tutorial

PlanAhead Design Tool

UG760 (v 13.4) January 18, 2012

XILINX®

Table of Contents

Tcl and SDC Tutorial

This tutorial shows you how to use the Xilinx® PlanAhead™ design tool to write scripts with the Tool Command Language (Tcl) API. It assumes that you are familiar with the PlanAhead tool Graphical User Interface (GUI) and project flows.

If you are not familiar with PlanAhead, follow the *Quick Front-to-Back Overview Tutorial (UG673)* before starting this tutorial.

The tutorial uses the features contained in the PlanAhead design product, which is bundled as a part of the ISE® Design Suite.

In this tutorial, you will:

- Use a sample design to explore the Tcl Console, the relationship between the GUI and the Tcl commands, and the log and journal files.

- Open a project in the PlanAhead tool using Tcl and batch scripts.

- Become familiar with the PlanAhead GUI, the Tcl Console, and online help for Tcl commands.

- Learn about the different modes of operating the PlanAhead tool.

- Run some of the basic built-in Tcl commands.

- Create batch-mode project creation and flow execution scripts.

- Explore Tcl objects, properties, and physical constraints.

- Perform a simple conversion of UCF timing constraints to Synopsys Design Constraint (SDC) equivalents and explored incremental static timing analysis reporting.

Software Requirements

The PlanAhead tool is installed with ISE Design Suite software. Before starting the tutorial, be sure that the PlanAhead tool is operational, and that the tutorial design data is installed.

For installation instructions and information, see the *ISE Design Suite: Installation and Licensing Guide (UG798)* at http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/iil.pdf.

Hardware Requirements

Xilinx recommends a minimum of 2 GB of RAM when using the PlanAhead tool on larger devices. For this tutorial, a smaller xc6vlx75t design is used, and the number of designs open at one time is limited. Although 1 GB is sufficient, it can impact performance.

Tutorial Design Description

The sample design used in this tutorial consists of a typical system on a chip design with a RISC CPU core connected to several peripheral cores using a Wishbone bus arbiter. The design targets an xc6vlx75Tff784 device. This tutorial uses a project file, which has already synthesized the HDL and is ready to be used.

Tcl Syntax

This tutorial contains code snippets that you can type into the PlanAhead tool at the Tcl Console. The Tcl syntax is shown with a prefixed right angle symbol > in bold, which is to prompt you to type specific commands into the Tcl Console window. An example is:

```
>get_cells cpuEngine
cpuEngine
```

The > character indicates the text to be typed, and below it is the expected response.

Step 1: Opening a Project

The PlanAhead tool lets you create several types of projects depending on where you are in the design flow. Register Transfer Level (RTL) sources or synthesized netlists can be used to create a project for development, analysis, implementation and bit file creation. This tutorial uses a synthesized netlist project that is not yet implemented.

Opening the PlanAhead Software

Open the PlanAhead tool:

- On Windows, select the Xilinx PlanAhead 13.4 desktop icon, or select Start > All Programs > Xilinx ISE Design Suite 13.4 > PlanAhead > PlanAhead.

- On Linux, go to some desired location and type "planAhead"

The PlanAhead Getting Started Help page opens.

Opening the Example Project

During this tutorial, you modify the example design project netlist. In this section, open the example design and then save the project to a different name so the original example design project can be re-used.

1. In the Getting Started page, select **Open Example Project > CPU (Synthesized)**.

2. Select **File > Save Project As** to save the project to a different project name

   The Save Project As dialog box opens, as shown in the following figure.
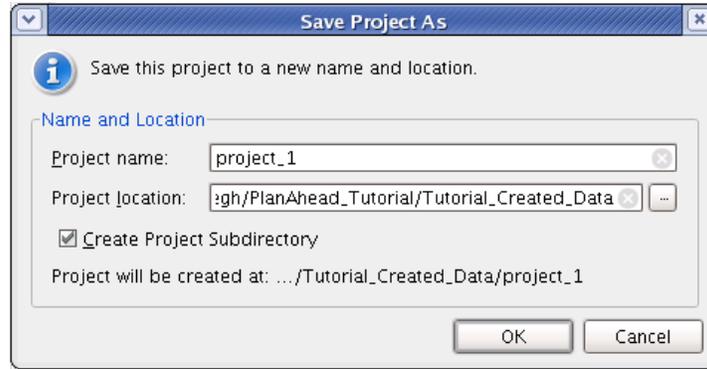
Figure 1: Save Project As Dialog Box

3. In the Project Name text box, enter a unique name for the project, such as "project_1".

4. Enter a unique Project location.

5. Click **OK**.

   The Project Manager opens with the design sources displayed in the Sources view.

6. In the Flow Navigator on the left side of the PlanAhead environment, click **Netlist Design**.

   The Netlist Design opens, as shown in the following figure, and is ready to explore.
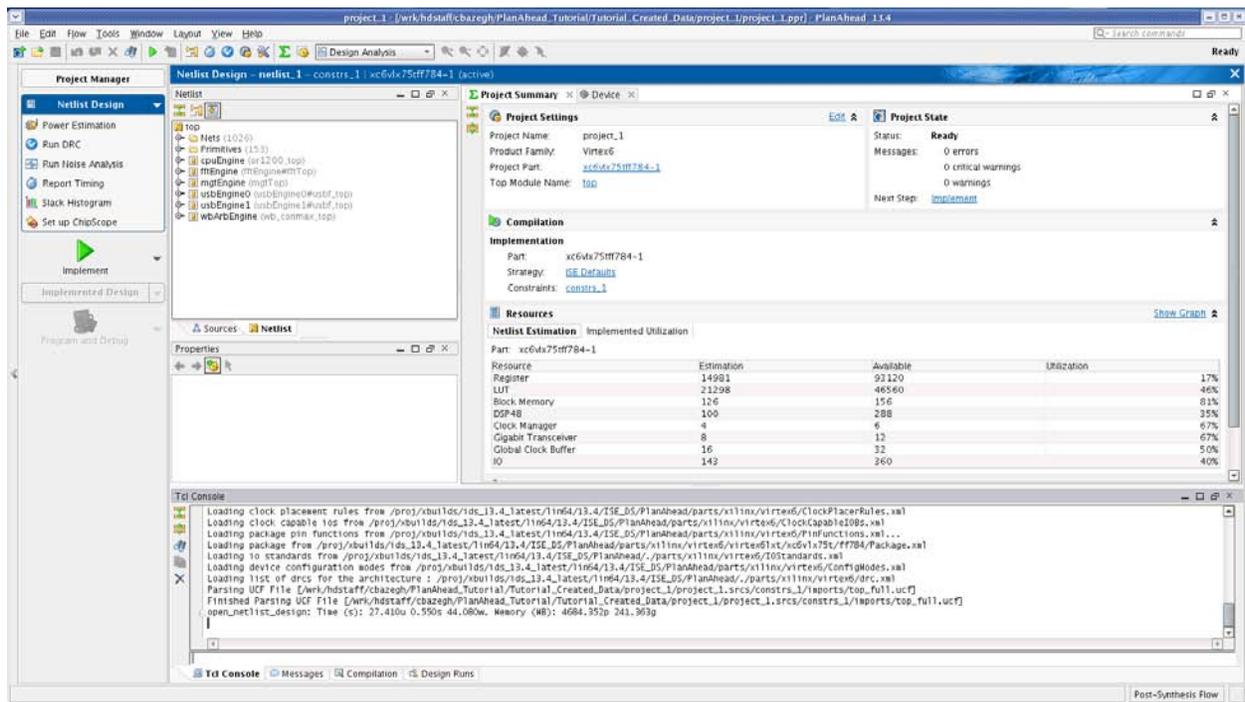
Figure 2: The Project in the Netlist Design Environment

Step 2: Exploring the Tcl Console and Online Help

When you start the PlanAhead tool in GUI mode, the main window displays the Tcl Console, which contains messages resulting from operations performed in the GUI. You can type Tcl commands directly into the text box below the Tcl Console. The Tcl Console contains a scrollable history of the software session also, shown in the following figure.
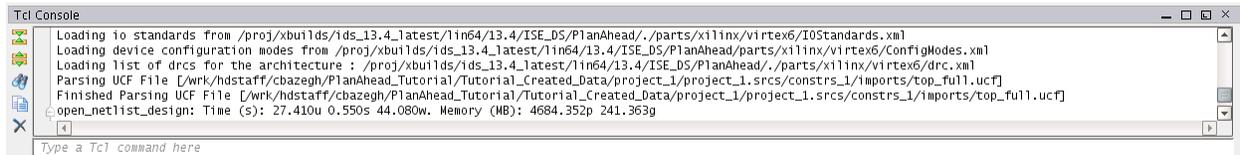
Figure 3: The PlanAhead Tcl Console

Viewing the Tcl Console and Messages

Type the following Tcl command in the Tcl Console text box.

```
>puts "Hello!"
```

The output is printed in the Console history.

```
Hello!
```

In this case, the built-in Tcl "puts" command prints string messages to the Console and to the log files for information purposes.

Examining Online Help

1. Type the following Tcl command in the Tcl Console text box:

```
>help
```

A list of command categories display.

2. For a list of the commands for a given category, type:

```
help -category <category name>
```

3. For a complete list of PlanAhead tool Tcl commands to be printed to the screen along with a brief description of each command, type:

```
>help *
```

Getting Help for a Specific Command

1. In the Tcl Console text box, type in the following:

    >create_project -help

    Note: You can also type "help create_project".

The complete help syntax for the **create_project** command is printed. Here is the first part of this:

```
Description:
Create a new project

Syntax:
create_project  [-part <arg>] [-force] [-quiet] <name> [<dir>]

Returns:
new project object

Usage:
  Name     Optional  Default  Description
  ------------------------------------
  -part    yes                Target part
  -force   yes                Overwrite existing project directory
  -quiet   yes                Ignore command errors
  <name>   no                 Project name
  <dir>    yes        .       Directory where the project file is saved
```

2. Type the following invalid command into the Tcl Console.

    >junk
    invalid command name "junk"

The Tcl interpreter issues an error.

> **Note**: Notice that a small red bar was placed next to the scroll bar to the right of the Tcl Console history to indicate an error occurred at this line. You can use this feature to scroll back through the command history and quickly see if warnings or errors occurred in the context of any messages. Warnings are colored yellow and errors are red.

When you type commands into the Tcl Console, the Tcl interpreter looks for known commands and defined procedures. If none are found that match the command, then it sends the command to the OS shell for execution. If no known command is found, then an error is issued.

Typing an OS Shell Command

In the Tcl Console, type:

```
>dir (Windows)
>ls (Linux)
```

The complete list of files in the current working directory displays. You can use this feature to access OS-specific commands from within the interpreter, or alternatively, you can use the Tcl built-in exec command. Type "help exec" in the Tcl Console for more information on this command.

Examining the Journal and Log Files

Each time you launch the PlanAhead tool, it creates two files that are very useful for understanding the creation of Tcl scripts.

*   Journal file (planAhead.jou)—The journal file contains the history of all commands executed in your session, which include commands entered in the Tcl Console, commands sourced from a script, and commands run in the GUI that have a Tcl-command equivalent.

- Log file (`planAhead.log`)—The log file contains all the journal commands, and the information, warning, and error messages created from running the commands. This file provides the context for each executed command.

Use the journal file for learning Tcl syntax for a given command. For example, you can use a GUI command, and then look at this file for the expected Tcl syntax for the operation that was performed.

Note: The journal and log files are located in the "start-in" directory, which is the current working directory where you invoked the PlanAhead executable for Linux.

For Windows, this directory is defined by the `%APPDATA%` environment variable, which is normally mapped to the following location:

- In Windows XP—`C:\Documents and Settings\`*<user_name>*`\Application Data\Xilinx\PlanAhead`

- In Windows 7—`C:\Users\`*<user_name>*`\AppData\Roaming\Xilinx\PlanAhead`

1. Referring to the location specified above, view the `planAhead.log` and `planAhead.jou` files.

2. From within PlanAhead you can access either file via File→Open Log/Journal File menu.

3. Look for the commands you executed above, and any information, warning and error messages.

   Note: Be aware that each time you launch the PlanAhead tool, it overwrites the journal and log files. Keep this in mind if you want to save these files for future reference. When you launch the PlanAhead tool, a backup copy of your last set of files is saved to `.jou_backup` and `.log_backup`.

Step 3: Running Tcl Scripts

You have been working thus far with the PlanAhead tool GUI mode. There are three operating modes: GUI, Interactive Shell, and Batch Mode.

This section explores the different modes for executing Tcl commands and scripts.

Executing a Tcl Script in GUI Mode

In the default GUI mode, Tcl commands are typed directly in the Tcl Console, or are sourced from the Tools menu command.

1. Create a file called `step3.tcl` in any text editor, such as Notepad, EMACS, or VI, and enter the following command:

   ```
   puts "Hello World!"
   ```

2. In the GUI, click **Tools > Run Tcl Script** to launch the Run Script dialog box, as shown in the following figure.
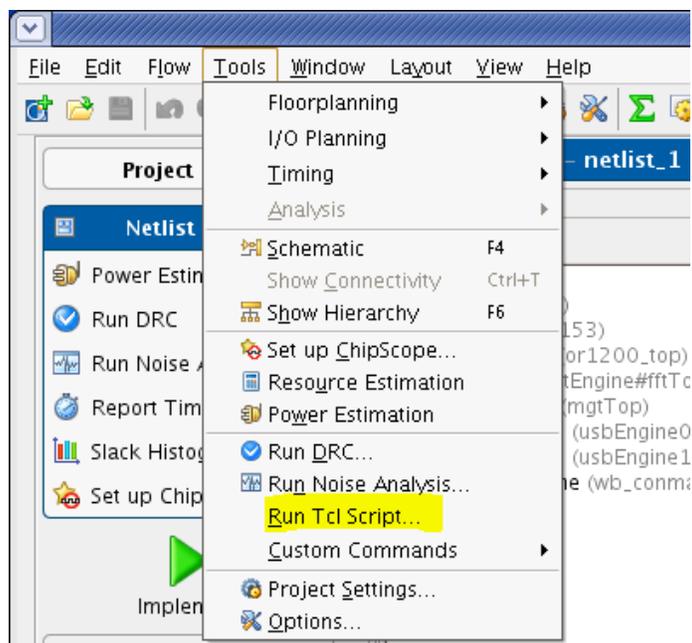
Figure 4: Launching the Run Tcl Script Command

3. Navigate to the directory where you created the file **step3.tcl**.

4. Select the file.
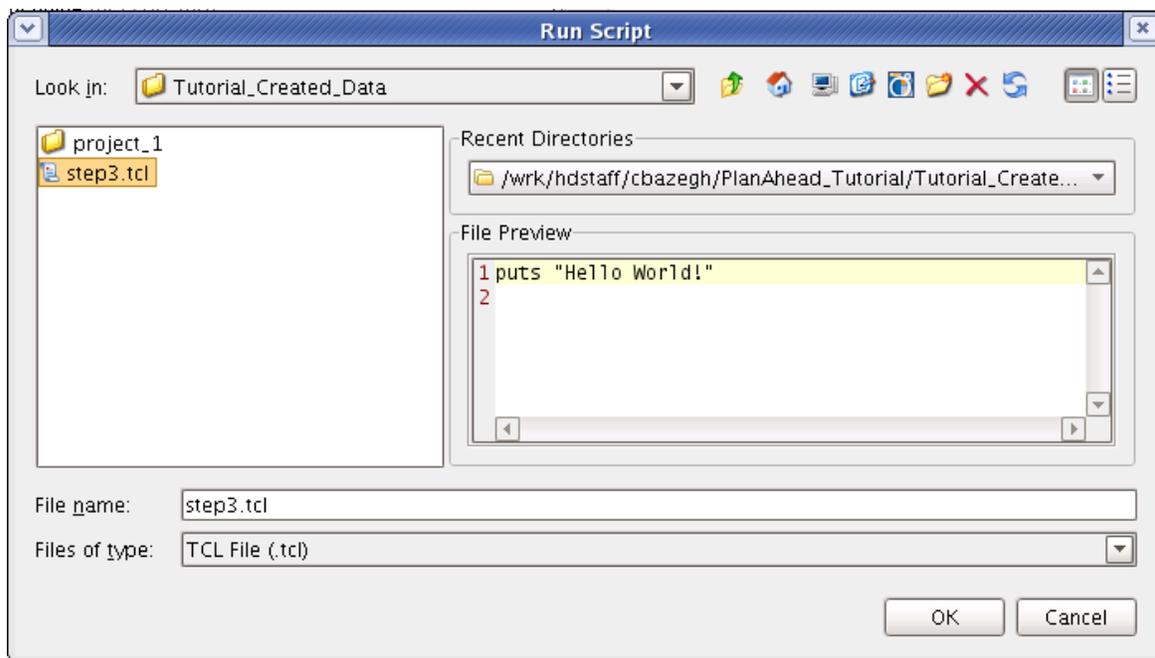
5. Click **OK**.

Figure 5: Previewing and Selecting Tcl Script File

"Hello World!" prints in the Tcl Console.

In addition to invoking Tcl scripts from the Tools command, you can also run the script using the Tcl Console by typing:

```
>source <path_to_file>/step3.tcl
```

6. Exit the PlanAhead tool by selecting **File > Exit** or typing exit in the Tcl Console.

Sourcing a Script from the Command Line

You can run Tcl commands when you launch the PlanAhead tool. Enter the **?source** command line option at a DOS terminal (cmd) or Linux shell prompt from the directory you saved your file.

1. Launch the PlanAhead tool and run the step3.tcl file that you already created.

```
>planAhead -source step3.tcl
```

This launches the GUI and sources the specified Tcl script. When the launch completes, you can run commands in the GUI or enter Tcl commands in the Tcl Console.

Note: This assumes the PlanAhead executable is installed and resides in your command search path (`$PATH` in Linux, and `%Path%` in Windows).

If the PlanAhead tool GUI is not accessible, you can use any of the following options:

- Source the settings32.bat or settings64.bat (or .sh for Linux) environment setup script provided with the ISE installation.

- Prefix the PlanAhead tool command with the fully qualified path to the installation directory for the PlanAhead tool.

- Modify the environment variable PATH to make sure the <PlanAhead_install_dir>/PlanAhead/bin directory is in the executable search path.

Note: You should see in the Tcl Console the sourcing of your Tcl script and the result.

2. Select **File > Exit** or type **exit** in the Tcl Console to close the PlanAhead tool.

Using the Interactive Shell

The PlanAhead tool offers an interactive Tcl shell mode for non-GUI interactive sessions.

1. Run the interactive shell mode use the ?mode switch:

```
>planAhead -mode tcl
```

The PlanAhead shell prompt (PlanAhead%) displays and is ready for you to type commands directly into the tool, without the GUI. All Tcl commands are available "live," and you can run commands interactively, and query projects and designs as you would in the GUI.

2. Exit the PlanAhead tool, by typing the following in the Tcl Console.

```
>exit
```

Another option to the interactive shell mode is to run the PlanAhead tool in Tcl mode and source a starting script.

3. Run the software in Tcl mode and source a script.

```
>planAhead -mode tcl ?source step3.tcl
```

4. Type in the help command to view command help in the interactive shell.

```
>help
```

5. Exit the PlanAhead tool.

    >exit

Using the Batch Mode Command

The batch mode is similar to the interactive mode in that it does not invoke the GUI, but requires the –source option with a valid Tcl script. Batch mode runs the Tcl script and then exits.

Run the step3.tcl file by sourcing it in batch mode, as follows:

    >planAhead -mode batch -source step3.tcl

This launches the PlanAhead tool. It runs the script, then quits without offering any interactive shell prompt to type commands.

Step 4: Exploring Basic Tcl Built-In Commands and Syntax

This tutorial does not provide a complete background on all of the capabilities of Tcl. See the Tcl website (http://www.tcl.tk/doc/) for Tcl fundamentals:

- Tcl reference information

- An introductory tutorial to Tcl 8.5.

- Documentation for built-in commands that are not PlanAhead tool-specific.

Note: PlanAhead has integrated the latest release of Tcl, version 8.5.

Understand the Tcl Interpreter and Basic Syntax

Tcl is an interpreted scripting language, there is no need to compile Tcl scripts to machine language or assembly code. There is an interpreter, which takes Tcl commands as input and evaluates them according to the language semantics. The interpreted nature of the language makes Tcl a very powerful interface to EDA tools; it allows you to type in commands that ask questions of the design, to which the tool responds with an answer.

In the GUI, the Tcl Console and the "planAhead%" prompt in the interactive shell mode are the interface to the interpreter. You can type commands at the prompt and those commands are passed to the interpreter for evaluation. You have been interacting with the interpreter in the previous sections of this tutorial.

The basic syntax for Tcl commands is:

    <command> <options>

Commands are evaluated sequentially (if in a script with multiple commands), and the command is evaluated left to right.

Understanding Variables and Substitutions

You can create variables in Tcl with the set command. The Tcl language is loosely typed which means you can create variables that hold data without concern about what type of data the variable contains.

Now you will do some Tcl programming.

1. Launch PlanAhead in Tcl mode (>planAhead -mode tcl) or in a Tcl shell if you have that installed (tclsh).

2. Create a simple string variable by typing:

```
>set var1 "Hello World!"
Hello World!
```

The string "Hello World!" echoes to the Console history.

3. Set variables to strings, integers, or floating point numbers with the set command as follows:

```
>set var2 1
1
```

The value for var2 is set to 1.

```
>set var3 3.14
3.14
```

The value for var3 is set to 3.14.

4. Set new variables using other variables using the concept of substitution as follows:

```
>set var4 $var1
Hello World!
```

The $ character tells the interpreter to look for a variable with the given name and substitute its value before doing the assignment to var4. The $ character is a special character in Tcl, which is important to note. A list of the special Tcl characters follows.

Character	Name	Behavior
[]	Square Brackets	Nests one command into another
{}	Curly Brackets	Literal string
""	Double Quotes	String that allows variable and command substitution
\	Backslash or Escape Character	Use before special characters so the interpreter does not attempt to read
;	Semicolon	End of a command
#	Pound Sign	Comment
$	Dollar Sign	Indicates the use of a previously defined variable.

To prevent substitution you can use either of the following mechanisms:

Literal Strings with curly braces {}

Back-slashes \ before special characters

5. Set a variable with the character for a dollar sign in it without the interpreter trying to substitute a variable, using either:

```
>set var5 {$var4}
```

or,

```
>set var5 \$var4
$var4
```

About Conditional Statements

Tcl supports conditional execution with **if** statements. To see how this works:

Type the following into the Tcl Console:

```
>if {$var2 == 1} {puts "Yay!"}
Yay!
```

In this example, the curly braces {} indicate the "body" of other statements. The interpreter tests to see if the variable named var2 equals 1, which it does, and it executes all of the commands in the second set of braces.

Note: You need to have a space between the two sets of {} or you will get the message "extra characters after close-brace".

Adding Additional Nested ifs With the elseif Command

The **elseif** command lets you nest multiple else conditions for an if/else conditional statement. You can string a number of elseif statements together to create a cascade of conditionals.

In the following example, the first line implements a three state conditional in which Yay! is output if the variable var2 equals 2, Nay! is output if var2 is 3.14, and Maybe! is output if either preceding condition is not met.

Type the following into the Tcl Console:

```
>if {$var2 == 2} {puts "Yay!"} elseif {$var3 == 3.14} {puts "Nay!"} else {puts
"Maybe!"}
Nay!
```

About Lists and Looping

In the same way that Tcl variables are loosely typed, there is a notion of container variables that hold more than one value. Lists are also similarly loose and can be as simple as a string with values separated by spaces. The following table contains examples of Tcl variables.

Command	Returns	Description
>set colors "red blue green"	red blue green	Creates a variable called "colors" and assigns it a string, which is a list.
>set colors2 [list brown black white]	brown black white	Use the explicit list command to build up a list.
>llength $colors	3	Returns the number of items in the $colors list.
>lindex $colors 0	red	Returns the value of the first item in the $colors list.
>lindex $colors2 end	white	Returns the value of the last item in the $colors2 list.

The most common way to iterate through a list is with the foreach command.

To operate on each of the colors in the colors list in table above, type the following in the Tcl Console:

```
>foreach c $colors {puts $c}
red
blue
green
```

This prints each color as it loops through the list.

About Nesting Commands

Nesting commands embeds multiple commands inside other commands using the square bracket special characters. Nested commands get executed by the interpreter from the inner-most scope of the commands outward. A common way to use nesting of commands is shown by the math `expr` command:

```
>set var7 [expr 1 + 1]
2
>set var8 [expr $var3 / 10]
0.314
>set var9 [expr [expr 2 * 3] + 1]
7
```

You can nest any command, and the square brackets tell the interpreter to evaluate the commands in the enclosed brackets and then substitute the resulting value into the next command.

Understanding Error Handling

Errors in Tcl scripts normally halt script execution. There are built-in mechanisms to handle error trapping, so that you can decide to continue execution. The catch command takes any command as an argument and returns 1 if there is an error:

1. Enter the following set of commands:

    ```
    >catch "junk" result
    1
    >puts $result
    invalid command name "junk"
    ```

    The results of a catch command can be combined with if statements to handle errors:

    ```
    >if {[catch "junk" result]} {puts "ERROR_IN_MY_SCRIPT!"}
    ERROR_IN_MY_SCRIPT!
    ```

    The catch command traps the error resulting from the unknown "junk" command and runs the commands in the block of code with the **puts** command in it.

2. Exit the PlanAhead tool or the Tcl shell.

    ```
    >exit
    ```

Step 5: Creating A New Project: Running Synthesis and Implementation

Basic project creation starts with the `create_project` command. There are a few options required when creating a project, which include:

- Name of the project

- Directory in which to create the project

- Default part the project will target

You will create a script to create of a small project and run RTL synthesis in batch mode.

Creating a Project

1. Create a new file named **step5.tcl**.

   In the next steps, you will add commands to this file.

2. Type (or copy) the following basic variable definitions in the step5.tcl file to use later. Be sure to modify the <Install_Dir> to the correct path for your ISE installation (c:/Xilinx13.4/ISE_DS):

   ```
   set projDir [file dirname [info script]]
   set srcDir  <Install_Dir>/PlanAhead/testcases/PlanAhead_Tutorial/Sources/hdl/
   set projName usbf
   set topName usbf_top
   set device xc6vlx75tff484-1
   ```

   In these commands:

   - The [info script] command returns the full file name of the script executed by the Tcl interpreter.

   - The file name is passed to the file `dirname` command, which returns the directory in which the script is located. You are creating the project in the same directory location in which the script resides.

3. Add the following to the script to check to see whether a directory already exists, and if it does, remove it so there is a clean run:

   ```
   if {[file exists $projDir/$projName]} {
      # if the project directory exists, delete it and create a new one
      file delete -force $projDir/$projName
   }
   ```

4. Use the **create_project** command to create a new project, using the variables created above for the project name, directory and the target part, and type the following:

   ```
   create_project $projName $projDir/$projName -part $device
   ```

5. Set the **design_mode** property on the source set:

   ```
   set_property design_mode RTL [get_filesets sources_1]
   ```

   This will set the project to be an RTL project, which is a container object for all the RTL source files. If you were creating a netlist-based project, based on a post-synthesis netlist, this property value would be Netlist instead of RTL.

6. Define the RTL sources to add to the project by typing the following:

   ```
   set verilogSources [glob $srcDir/FifoBuffer.v $srcDir/async_fifo.v $srcDir/rtlRam.v
   $srcDir/$projName/*.v]
   ```

Tcl provides a built-in command to query all files that match a wildcard search, called **glob**. Type **help glob** in the Tcl Console for more information on this command.

7. Use the **import_files** command to add the source files to the project and copy them locally, by typing the following:

    ```
    import_files -fileset [get_filesets sources_1] -force -norecurse $verilogSources
    ```

    This command string imports the individual files into the project and puts those files in container objects called *filesets*. The default fileset is named **sources_1**. The fileset preserves these files.

    In this command:

    - The -force option overwrites any previous sources by the same name.

    - The -norecurse option instructs the PlanAhead tool not to recursively search every subdirectory and add any additional files it finds.

8. Set the name of the top-level module or entity in order to synthesize the top level:

    ```
    set_property top $topName [get_property srcset [current_run]]
    ```

9. Save the file, and run the PlanAhead tool sourcing the step5.tcl script you just created:

    ```
    >planAhead -mode batch -source step5.tcl
    ```

This launches the software and creates the project based on the script and exits when completed. You should now see a directory named "usbf" from where you launched PlanAhead.

Adding Commands to Synthesize the Project

In the previous section, you created a script to create an RTL project. In this procedure, you will add commands to perform synthesis using the default synthesis strategy.

1. Open the `step5.tcl` in a text editor, and add the following commands to the end of the script, after the `set_property` top command:

    ```
    launch_runs -runs synth_1
    ```

    In the PlanAhead GUI when you launch a synthesis or implementation run, you are launching the process in a separate thread, so that you can continue to use the GUI to analyze your design.

    Tcl is the same, and without doing anything special, synthesis would run. Because you are running in batch mode, you must enter commands to block further execution until the synthesis run completes. This ensures that the next step, which is running implementation, works correctly.

2. Enter the command to block execution as follows:

    ```
    wait_on_run synth_1
    ```

3. Save the `step5.tcl` file.

4. Run the PlanAhead tool and source the script you just created:

    ```
    >planAhead -mode batch -source step5.tcl
    ```

    The PlanAhead tool deletes the previous project, re-creates it again, and runs through synthesis using the default synthesis strategy. The PlanAhead tool provides a number of built-in strategies for synthesis. Choosing a different one is as simple as setting the strategy property on the synthesis run, and recompiling.

5. Add the following command to the `step5.tcl` file before the `launch_runs synth_1` command:

   `set_property strategy PowerOptimization [get_runs synth_1]`

   This chooses the power optimization strategy.

6. Save the `step5.tcl` file.

7. Run the PlanAhead tool and source the script you just created:

   `>planAhead -mode batch -source step5.tcl`

The PlanAhead tool deletes the prior project, creates a new one, and synthesizes using the power optimization strategy.

Launching Implementation

In the previous steps, you created a script to create a project and synthesize it using XST.

Next, you will add commands to launch implementation, and run the script.

1. Open `step5.tcl` in a text editor, and add the following commands to the end of the script, after the `wait_on_run synth_1` command:

   `launch_runs -runs impl_1`
   `wait_on_run impl_1`

   Similar to the synthesis run, these two commands launch the implementation run using the default strategy and block until completion.

2. Save the `step5.tcl` file.

3. Run the PlanAhead tool and source the script you just created:

   `>planAhead -mode batch -source step5.tcl`

   This script deletes the previous project, recreates a new one, and runs through implementation with the default strategy.

4. To use a different implementation strategy, such as the timing-driven map flow, add the following command before the "launch_runs -runs impl_1" command:

   `set_property strategy MapTiming [get_runs impl_1]`

5. Save the `step5.tcl` file.

6. Run the PlanAhead tool and source the script you just created:

   `>planAhead -mode batch -source step5.tcl`

This script deletes the previous project, recreates a new one, and runs through implementation with the `MapTiming` flow.

Opening the Implementation Results

After a synthesis or implementation run is complete, open the Netlist Design or Implemented Design to load it into memory and make it active in the PlanAhead tool. Following that you can perform further Tcl commands and operations on the active design.

1. To open the post-implementation netlist, add the following to the end of step5.tcl:

   `open_impl_design`

2. Save the step5.tcl file.

3. Run the PlanAhead tool in Tcl mode and source the script you just created:

   ```
   >planAhead -mode tcl -source step5.tcl
   ```

4. When the `PlanAhead%` prompt appears, start the GUI by typing the following:

   ```
   PlanAhead% start_gui
   ```

   The script deletes the previous project, recreates a new one, runs through implementation flow, and then opens the implementation result design for further analysis and operation. Next, you launched the GUI for further interaction.

   Note: The **start_gui** command can also be added to the end of the step5.tcl script if you always want to load the GUI after processing is complete.

5. Exit PlanAhead.

   ```
   >exit
   ```

Step 6: Exploring Netlist Objects, Properties, and Physical Constraints

In the previous step, you created a project and executed a simple flow using Tcl in batch mode. In this step, you will explore more of the netlist access commands using core SDC commands.

Opening the Project

1. In a text editor, create a file called `step6.tcl` in the same directory as you used before and type (or copy and paste) the following commands into the file. Remember to replace `<Extract_Dir>` with the correct path.

   ```
   set projDir [file dirname [info script]]
   set srcDir <Extract_Dir>/PlanAhead_Tutorial/Projects/
   set projName project_cpu_netlist
   set topName top
   set device xc6vlx75tff484-1
   # open existing project
   open_project $srcDir/$projName/$projName.ppr
   # now open the post-synthesis netlist design
   open_netlist_design
   ```

2. Save `step6.tcl` file.

3. Run the PlanAhead tool and source the script file:

   ```
   >planAhead -source step6.tcl
   ```

This launches the PlanAhead tool, opens the project, and loads the netlist design so that it is ready to accept further commands.

Exploring Objects and Properties

First, explore a few of the commonly used object types in the PlanAhead tool GUI. For flow control it is helpful to review properties of run objects. This is useful for querying projects to determine their current status. Maximize the Tcl Console to see the results better.

1. Enter the following commands in the Tcl Console to explore properties of run objects:

   ```
   >set runList [get_runs]
   >report_property [lindex $runList 0]
   >get_property status [lindex $runList 0]
   ```

These commands illustrate the properties of runs that can be queried through Tcl. The status of the implementation run impl_1 should be "Not Started" since we have not yet run implementation.

The most commonly used of the core SDC commands is the **get_cells** command, which provides a way to query instances in the netlist design by name. The -hierarchical switch instructs the PlanAhead tool to apply the pattern supplied at each level of hierarchy in the design. The below command has the effect of returning every cell in the design.

2. Type the following commands in the Tcl Console to see properties of cell objects:

```
>set cellList [get_cells -hierarchical *]
>report_property [lindex $cellList end]
>get_property lib_cell [lindex $cellList end]
```

These commands query all cells in the design, and print a property report on the last cell in the list. The final command returns the value of the primitive library cell property.

The **get_nets** command is another commonly used object query command, which takes a hierarchical search pattern also.

3. Type the following commands in the Tcl Console to experiment with net object properties:

```
>set netList [get_nets *]
>report_property [lindex $netList 0]
>get_property type [lindex $netList 0]
```

These commands query all the net objects at the top-level of the hierarchy, and print a report of the property values for the first one in the list. Finally, the type property is queried, which should return a Signal type.

Port objects are queries with the **get_ports** command.

4. Type the following commands to experiment with port objects and their properties:

```
>set portList [get_ports *]
>report_property [lindex $portList 0]
>get_property iostandard [lindex $portList 0]
```

These commands return all the top-level ports in the design, and print out a list of usable properties on each for scripting. Finally, the I/O standard applied to a port can be queried with the get_property command, which should return LVCMOS25.

Pin objects can be queried with the **get_pins** command.

5. Type the following commands in the Tcl Console to explore pin object properties:

```
>set pin [get_pins OpMode_pad_0_o_0/D]
>report_property $pin
>get_property setup_slack $pin
```

These commands query a specific pin, the D input to a flop, and report all the properties available on pin objects. Pin objects are tightly coupled to the static timing analysis engine, and you can query pins based on setup or hold slack properties. Querying timing analysis related properties will invoke the timing analysis engine, which will build a timing graph. This is a useful debug and analysis feature.

Filtering Object Queries With Properties

You can combine properties with the **-filter** option to any **get_** command to filter out the list of returned objects based on specific criteria. Below are some examples of using object access commands with filtering.

1. Type the following commands in the Tcl Console:

```
>get_nets -filter {type == "Global Clock"}
>set cellList [get_cells * -hierarchical -filter "lib_cell =~ FD*"]
```

The first command queries global clock nets and the second command returns a list of all cells in the design where the primitive type matches a string pattern that starts with FD. This returns all flip flops in a design, such as FDR primitives. Object `lib_cell` properties map directly to UNISIM primitives.

Objects are related to one another through netlist connectivity: ports connect to nets; nets connect to pins; and pins connect to cells.

The `-of_objects` option to the object access commands are an important way to traverse netlist objects. The can be abbreviated to `-of` because Tcl auto completes when there is no ambiguity about the command.

2. To explore these commands, type the following commands in the Tcl Console:

```
>set cell [get_cells fftEngine/control_reg_1]
>set pin [lindex [get_pins -of_objects $cell -filter {direction == IN}] 1]
>set net [get_nets -of_objects $pin]
>set driver [get_pins -of_objects $net -filter {direction == OUT}]
>set driverCell [get_cells -of_objects $driver]
```

These commands query connectivity relationships and traverse the netlist based on a variety of properties and connectivity information. This is a powerful capability.

Exploring the Physical Constraints

Exploring physical constraints is useful, but you need to be able to set physical constraint values in Tcl in a manner that is consistent with UCF. The following are command examples of setting physical constraints through Tcl instead of UCF.

1. Type the following commands in the Tcl Console to see the physical constraints being applied:

```
>set_property IOSTANDARD SSTL15 [get_ports cpuClk]
>set_property site IOB_X1Y72 [get_ports cpuClk]
>set_property is_fixed true [get_ports cpuClk]
>set_property loc SLICE_X0Y73 [get_cells fftEngine/control_reg_1]
>set_property bel AFF [get_cells fftEngine/control_reg_1]
>set_property is_fixed true [get_cells fftEngine/control_reg_1]
```

These commands set physical constraints such as IOSTANDARDS, LOC, and BEL constraints. Most attributes that can be applied in HDL would propagate to attributes in an EDIF netlist, which can be queried with `get_property` and can be set with `set_property`. To see all the properties for an object you can use the `report_property` command.

2. Close the PlanAhead tool:

- In the Tcl Console, type **exit**.

- Select **File > Exit**.

- In the main window, click the **X** in the upper right hand corner.

Step 7: Using Static Timing Analysis with Tcl and SDC

The PlanAhead tool has a Static Timing Analysis (STA) engine that is separate from TRACE, the ISE STA engine. The PlanAhead tool STA engine is compatible with Synopsys Design Constraints (SDC) and supports incremental timing analysis.

In this step, you convert a simple UCF timing constraint file to SDC. SDC is currently only supported by the PlanAhead tool, and timing constraints do not port forward to ISE implementation and timing analysis tools. However, SDC is a powerful analysis and debug tool for netlist exploration. SDC is used primarily for timing constraints, so in this section you will explore some of the basics of SDC, specifically clocking, I/O constraints, and exceptions.

Setting up a UCF Conversion

In this section, you will open one of the sample projects, rename the project, save it with a different name, and disable UCF based timing in preparation for applying SDC constraints.

1. In a text editor, create a file called `step7.tcl` in the same directory you used before, and type (or copy and paste) the following commands into the file. Be sure to replace the *<Extract_Dir>* with the appropriate path for the `$srcDir` variable.

    ```
    set projDir [file dirname [info script]]
    set srcDir <Extract_Dir>/PlanAhead_Tutorial/Projects/
    set projName project_cpu_netlist
    set topName top
    set device xc6vlx75tff484-1

    # open project
    open_project $srcDir/$projName/$projName.ppr

    # rename the project
    set projName ${projName}SDC

    # save it to a new name and location
    if {[file exists $projDir/$projName]} {
    # if the project dir exists, delete it and create a new clean one
    file delete -force $projDir/$projName}
    save_project_as $projName $projDir/$projName
    # now disable all ucf files - in preparation for SDC
    set_property is_enabled false [get_files *.ucf]

    # Create a new constraint set for XDC and make it active
    create_fileset -constrset constrs_3
    set_property constrs_type XDC [get_filesets constrs_3]
    set_property constrset constrs_3 [get_runs impl_1]
    # now open the implementation results
    open_netlist_design
    ```

2. Save `step7.tcl`.

3. Run the PlanAhead tool, and source the script file:

    ```
    >planAhead -source step7.tcl
    ```

This command opens the design, saves it with a new name, and brings up the GUI with the design open. Alternatively, you can open the project manually with the GUI and rename the project.

Managing Clocks

In this section, you will create a file, and add clocks. Later, you will add constraints and source the file repeatedly to debug timing constraints with incremental static timing analysis.

In this procedure, the following `UCF TIMESPEC` constraints are replaced with the equivalent SDC `create_clock` constraints.

```
# Timing Constraints:
TIMESPEC TS_cpuClk = PERIOD "cpuClk" 13 ns;
NET "cpuClk" TNM_NET = "cpuClk";

TIMESPEC TS_wbClk = PERIOD "wbClk" 9 ns;
NET "wbClk" TNM_NET = "wbClk";

TIMESPEC TS_usbClk = PERIOD "usbClk" 5.25 ns;
NET "usbClk" TNM_NET = "usbClk";

TIMESPEC TS_phy_clk_pad_0_i = PERIOD "phy_clk_pad_0_i" 11 ns;
NET "phy_clk_pad_0_i" TNM_NET = "phy_clk_pad_0_i";

TIMESPEC TS_phy_clk_pad_1_i = PERIOD "phy_clk_pad_1_i" 11 ns;
NET "phy_clk_pad_1_i" TNM_NET = "phy_clk_pad_1_i";

TIMESPEC TS_fftClk = PERIOD "fftClk" 7 ns;
NET "fftClk" TNM_NET = "fftClk";
```

Note: The TIMESPEC PERIOD constraint from UCF is equivalent to the SDC create_clock command.

1. Create a new file called **top.sdc** using a text editor.

2. Type the following into the top.sdc file:

   ```
   create_clock -name cpuClk -period 13 [get_ports cpuClk]
   ```

   This command creates a clock named cpuClk with a period of 13 ns, with a rising edge at 6.5 ns rooted on the top-level port named cpuClk.

3. Add in the other clocks by typing the following into the `top.sdc` file:

   ```
   create_clock -name wbClk -period 9 [get_ports wbClk]
   create_clock -name usbClk -period 5.25 [get_ports usbClk]
   create_clock -name phy_clk_0 -period 11 [get_ports phy_clk_pad_0_i]
   create_clock -name phy_clk_1 -period 11 [get_ports phy_clk_pad_1_i]
   create_clock -name fftClk -period 7 [get_ports fftClk]
   ```

4. Save **top.sdc** in the current working directory.

5. In the PlanAhead tool Tcl Console, source `top.sdc` to create the clocks.

   ```
   >source top.sdc
   ```

   If your working directory is different from the directory where you saved the `top.sdc` file, type the following instead.

   ```
   >source $projDir/top.sdc
   ```

   In the Tcl Console you see the `create_clock` commands as comments and the clock name created as a return from the last command, `fftClk`.

6. To see the timing results with your newly created clocks, run the `report_timing` command:

   ```
   >report_timing
   ```

The timing engine is invoked and the worst case timing path is reported by default.

Adding Input Constraints

Now, you will set up the input timing relationship that equates to the UCF OFFSET IN constraint. Here is an example:

```
NET "DataIn_pad_0_i[0]" OFFSET = IN 3 ns VALID 7 ns BEFORE "TS_usbClk" RISING;
```

1. In `top.sdc`, add the following statements to the end of the file:

    ```
    set_input_delay -clock phy_clk_0 -max 2.25 [get_ports {DataIn_pad_0_i[0]}]
    set_input_delay -add_delay -clock phy_clk_0 -min 4 [get_ports {DataIn_pad_0_i[0]}]
    ```

    The first constraint states that the DataIn_pad_0_i[0] signal arrives 2.25 ns after the rising edge of the phy_clk_0 clock domain. The second constraint provides the min-delay (hold analysis) equivalent to the VALID window on the OFFSET IN. The **-add_delay** option to the second constraint tells the engine to preserve the previously defined max delay when adding the min delay value.

2. Save top.sdc in the current working directory.

    The PlanAhead tool provides a command to delete all timing constraints, so you can source the same SDC file repeatedly while experimenting.

3. Type the following in the Tcl Console text box:

    ```
    >reset_timing
    >source top.sdc
    ```

4. To see the timing results with your clock, run the `report_timing` command:

    ```
    >report_timing -from [get_ports {DataIn_pad_0_i[0]}]
    ```

Looking at the timing path, you can see that the slack is 10.234 ns. The data path delay is 1.937 ns.

Remember that you defined the clock period as 11ns in the create_clock for phy_clk_0. You set the input delay on DataIn_pad_0_i[0] as 2.25 ns, which you can see on the line labeled Input Delay.

Adding Output Constraints

Next, you should constrain the output signal requirements according to the specification in the OFFSET OUT constraint. The equivalent in SDC is the `set_output_delay` command. Here is an example from UCF for this project:

```
NET "DataOut_pad_0_o[0]" OFFSET = OUT 4 ns AFTER "TS_usbClk" RISING;
```

1. At the end of the top.sdc file, add the following constraint:

    ```
    set_output_delay -clock wbClk 1.25 [get_ports {DataOut_pad_0_o[0]}]
    ```

2. Save the `top.sdc` file in the current working directory.

3. In the Tcl Console, reset timing and source top.sdc to recreate the constraints by entering the following in the Tcl command text box:

    ```
    >reset_timing; source top.sdc
    ```

    Note: The semicolon (;) in the previous command is a way to combine multiple commands on the same line. The semicolon instructs the Tcl interpreter to execute the `reset_timing` command first, and then source the `top.sdc` file.

4. To see the timing results with the new setting, run the `report_timing` command:

    ```
    >report_timing -to [get_ports {DataOut_pad_0_o[0]}]
    ```

Looking at the timing report, you can see the output delay you set of 1.25 ns on the line labeled Output Delay. The data path delay is 5.851 ns, which includes both logic and route delays. You defined the wbClk to have a period of 9 ns, which is listed as the Requirement.

Until this point, you have been deleting the timing graph (with `reset_timing`) and recreating the timing analysis graph each time you modified the `top.sdc` file. This is not always necessary. You can use incremental STA capabilities by typing constraints and exceptions directly in the Tcl Console.

Using Multi-Cycle Paths

Next, you will convert a multi-cycle path from UCF to the SDC equivalent. Multi-cycle paths are equivalent to the FROM/TO constraints in UCF with a TIMESPEC multiplier. The following is an example for this project:

```
# Multi-cycle paths for ALU:
NET "cpuEngine/or1200_cpu/or1200_alu/*" TPTHRU = "GRP_ALU_DATAOUT";
TIMESPEC TS_ALU_MCP = FROM "cpuClk" THRU "GRP_ALU_DATAOUT" TO "cpuClk" TS_cpuClk *
2;
```

1. To relax timing to flip-flops and give it two clock cycles to meet setup, use the SDC `set_multicycle_path` command.

2. Type the following in the Tcl Console:

```
>report_timing -through [get_pins cpuEngine/or1200_cpu/or1200_alu/*]
>set_multicycle_path -through [get_pins cpuEngine/or1200_cpu/or1200_alu/*] 2
>report_timing -through [get_pins cpuEngine/or1200_cpu/or1200_alu/*]
```

- The first command performs an incremental STA update to generate a timing report through a number of pins of the design. The worst case timing path has a slack of 4.205 ns.

- The second command sets the new constraint on the paths, specifying that 2 clock cycles are allowed for timing.

- The third command updates only the portions of the timing graph affected by the new constraint and performs another incremental STA update.

Notice the required time for this path shifted from 13 ns to 26 ns, giving this path 2 full clock periods, and we see a slack of 17.205 ns.

Using False Paths

Similar to multicycle paths, for false paths we can incrementally add the SDC equivalent of TIG constraints to the UCF. The command to do this is called `set_false_path`. Suppose for the sake of analysis you wished to ignore all paths to the output pins. No timing between clock domains is the default for timing performed by TRACE with UCF constraints. With SDC-based timing, the default assumption is that all clocks are related. To get the same behavior with SDC you must explicitly set false paths between unrelated clock domains.

1. Enter the following constraints directly into the Tcl Console:

```
>report_timing -from [get_clocks cpuClk]
>set_false_path -from [get_clocks cpuClk] -to [get_clocks wbClk]
>set_false_path -from [get_clocks wbClk] -to [get_clocks cpuClk]
>report_timing -from [get_clocks cpuClk]
```

The first report_timing command shows a failing path, but it is between the cpuClk and the wbClk, which is not a valid path. We set two false path commands between these clock domains and rerun timing. Notice that there is now no timing calculation done between the clock domains.

2. Exit the PlanAhead tool.

```
>exit
```

Note: As with all timing exceptions, you need to take care with false and multi-cycle path constraints. You must be certain that these commands are correct because you are overriding the default analysis. It is still possible to have a valid path violation that would be hidden because of a timing exception that was added.

Conclusion

In this tutorial, you:

- Used a sample design to explore the Tcl Console, and explored the relationship between the GUI, the Tcl commands, and the journal file.

- Learned to open a PlanAhead tool project using Tcl.

- Became familiar with the Tcl Console, and online help for Tcl commands.

- Learned about the different modes of operation in the PlanAhead tool.

- Explored some of the basics of Tcl built-in commands.

- Created batch-mode script to create a project creation and run a flow.

- Explored Tcl objects, properties, and physical constraints.

- Performed a simple conversion of UCF timing constraints to SDC equivalents, and explored incremental static timing analysis reporting.